PERFORMANCE ASSIGNMENT

Joseba Hernández Bravo and Jorge Vicente Puig

November 1

Performance optimization

1. **Which is the main explanation for the performance improvement due to loop fusion and the double buffer strategy?**

   The improvement obtained with both strategies is mainly explained due to the reduction of the instructions of our program that reduce the total number of machine instructions executed by the processor. In fact, with both strategies we are avoiding to traverse the matrix.

   In the loop fusion we are using another loop to made the traverse of the matrix while in the double buffer we are using an auxiliar pointer to swap the matrix adresses avoiding the time needed to copy the matrix element by element.

2. **Why is the loop interchange transformation improving performance so much?**

   The reason is due to the way the information of a matrix is stored in the memory. Because the information is stored by rows instead of columns, if we are able to make our program reading the matrix by rows we are saving a lot of computation time in accessing the memory.

3. **Why is the strength reduction transformation providing no performance improvement**

   When using the -*Ofast* option when compiling the program, it assumes automatically that fractions can be interpreted as multiplications. Thus, if we write "/4" it assumes that is equal to "$*0.25f$".

4. **Which percentage of the total time takes the execution of laplace_copy() on the version that includes both loop fusion and loop interchange? State some conclusion from this value.**

   The execution time for the program containing the modifications **Loop fusion** and **Loop inter** is equal to $4326,98[msec]$. If we modify it by removing the **laplace_copy()** function and using **Double buffer** we get a new runtime of $2629.76[msec]$. Therefore, the time needed to execute the function **laplace_copy()** is $1697,19[msec]$. Thus

   $$\%time\_laplace\_copy() = \frac{1697,19 \cdot 100}{4326.98} = 39.22\%, \tag{0.1}$$

   with this information we can assume that there is a big importance on the way we save the new information to the matrix.

5. **Explain the effect on performance (time and machine instructions) of the code motion modification.**

   The code motion modification is based on the fact that the square root function is monotonically increasing so the maximum of the square root of some numbers is the same as the square root of the maximum.

   Then we can calculate it at the end of the function instead of making the calculus on each iteration. In this way, we are saving a lot of unnecesary machine instructions and then reducing the computation time.

6. **Using the performance results obtained from the Differential Diagnostic experiment, which is the main performance bottleneck of the program lap2.c?**

   The main performance bottleneck of the program is the size or the dimension of the mesh.

   When the dimensions of the Matrixes are small, and therefore take up little space, the data is stored in the cache. Thus, the CPU has fast access to this data. However, as the dimensions of our array increase, most or all of the accesses will miss in the cache and will be solved by the main memory, increasing the latency of load and store instructions, reducing the IPC rate and delaying program's execution. +

7. **Explain the optimization idea implemented in lap2B.c and how much and why it works.**

The major difference between **lap2.c** and **lap2B.c** is the use of a two-dimensional TMP matrix instead of Anew. In the first of the two programs, all calculations performed on the matrix A are stored in the Anew matrix. In this way, at the end of each iteration we have two different matrices in memory. Once we have all the calculations done, the Anew matrix becomes A and so we continue with the next iteration.

In the second program, however, the problem is approached in a different way. In this new example, we use a two-dimensional matrix instead of Anew. Each time we perform the calculations for the row **A[j,:]** we save the results in the TMP matrix and then we continue with the next row. If you notice, when we get to the row **A[j+2,:]**, the only rows we need are **A[j+1,:]** , **A[j+2,:]** , **A[j+3,:]**, so we don't need row **A[j,:]** at all. It is then when we copy the information stored in the TMP matrix (**TMP[(j+2)%2:]**) into row **j** of the matrix A. With this procedure, we are using less memory than in the previous program and thus, we are reducing the execution time from $2.113, 72 msec$ to $1.840, 69 msec$ (using 100 iterations). Besides, while the first program makes 0.99 instructions per cycle, the second one makes 1.29.

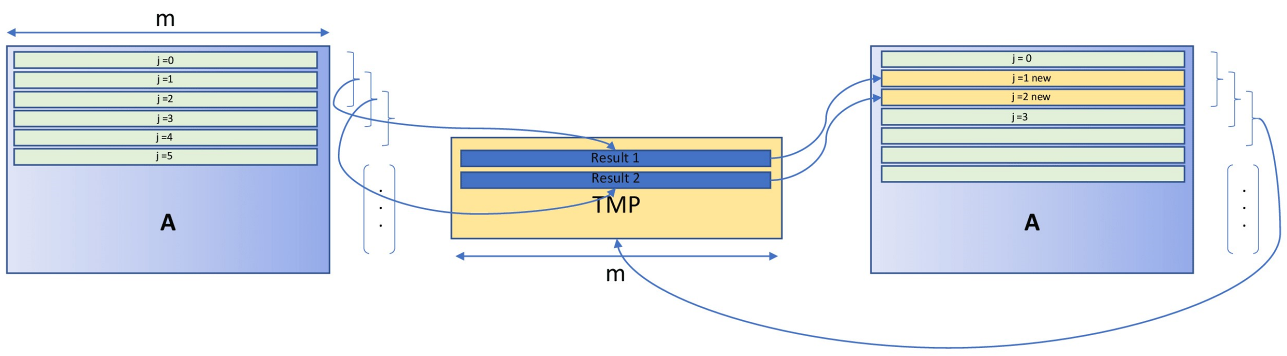

Figure 1: Graphical example of the algorithm of the lap2B.c program

8. **Explain the optimization idea implemented in lap2C.c and how much and why it works.**

In *lap2C.c* we are still working based on rows, as we did in *lap2B.c*, however we work with our matrix A and an auxiliar matrix t that has 3 rows.

The main idea is to store the errors calculated with the *Jacobi Method* in previous iterations on two rows of t while we are calculating the new iteration on the other row and then doing an iteration of *Jacobi method* on this errors to calculate the new values for the matrix A.

For an iteration we first calculate

$$t_{i,j} = \frac{a_{i,j-1} + a_{i-1,j} + a_{+1,j} + a_{i,j+1}}{4}$$

for 3 rows, and then we are able to calculate

$$a_{i,j} = \frac{t_{i,j-1} + t_{i-1,j} + t_{+1,j} + t_{i,j+1}}{4}$$

and we also store the computed error.

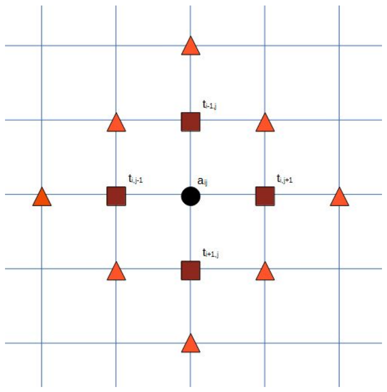

Figure 2: Iteration of the method for a_{ij}.

With this idea we are improving the optimization made on *lap2B.c*, beacuse we are avoiding to make the "copy line" step that we had to do.

In fact, we had to do 7.821.094.358 machine instructions while in this version only 5.915.416.503 and also we have an improvement on the execution time from 1.840,69*msec* to 1.295,98*msec.*

Notice that due to the bigger complexity of this method the instruccions per cycle from 1,29 in *lap2B.c* to 1,37 in *lap2C.c.*